- **You can explain the difference between Java Generics and C++ Templates**

- **You can implement simple generic functions**

- **You can implement variadic generic functions**

- **You can describe the type concepts required by a function template**

- **Recap Week 8**

- **Function Templates**

  - ◼ Motivation / Introduction

  - ◼ Type Concepts

  - ◼ Argument Deduction

  - ◼ Variadic Templates

  - ◼ Gotchas

Recap Week 8

- **Correctness**

  - It is much easier to use an algorithm correctly than implementing loops correctly

- **Readability**

  - Applying the correct algorithm expresses your intention much better than a loop

  - Someone else (or even you) will appreciate it when the code is readable and easily understandable

- **Performance**

  - Algorithms might perform better than handwritten loops

    - without sacrificing the "Readability" of your code

- **Binary arithmetic and logical**

  - `plus<> (+)`

  - `minus<> (-)`

  - `divides<> (/)`

  - `multiplies<> (*)`

  - `modulus<> (%)`

  - `logical_and<> (&&)`

  - `logical_or<> (||)`

- **unary**

  - `negate<> (-)`

  - `logical_not<> (!)`

- **binary comparison**
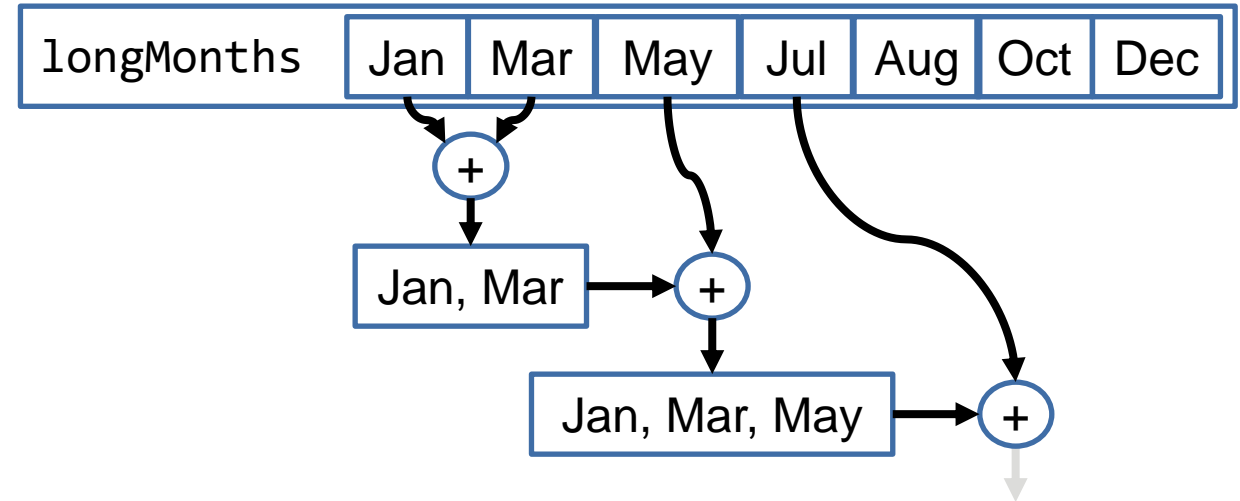
  - `less<> (<)`

  - `less_equal<> (<=)`

  - `equal_to<> (==)`

  - `greater_equal<> (>=)`

  - `greater<> (>)`

  - `not_equal_to<> (!=)`

```
transform(v.begin(),v.end(),v.begin(), v.begin(),std::multiplies<>{});
```

● **Some numeric algorithms can be used in non-numeric contexts**

● **Example `std::accumulate`**

- Sums elements that are addable (+ operator)

- Or based on a custom binary function

- Returns the "sum" (accumulated value)

| longMonths | Jan | Mar | May | Jul | Aug | Oct | Dec |

```
std::vector<std::string> longMonths{"Jan", "Mar", "May", "Jul", "Aug", "Oct", "Dec"};
std::string accumulatedString = std::accumulate(
        next(begin(longMonths)), //Second element
        end(longMonths),         //End
        longMonths.at(0),        //First element, usually the neutral element
        [](std::string const & acc, std::string const & element) {
                return acc + ", " + element;
        }); //Jan, Mar, May, Jul, Aug, Oct, Dec
```

- **If you use an iterator for specifying the output of an algorithm you, need to make sure that enough space is allocated**

```cpp
std::set<unsigned> numbers{1, 2, 3, 4, 5, 6, 7, 8, 9};
std::vector<unsigned> primes{};
auto isPrime = [](unsigned u) {/*...*/};
std::copy_if(begin(numbers), end(numbers), begin(primes), isPrime);
```

- **Vector `primes` is empty, which will not be changed by the iterator passed to copy**

numbers		1	2	3	4	5	6	7	8	9

- **The output is copied into (possibly) unallocated memory**

primes		2	3	5	7

Unallocated memory

DANGER

**Undefined
Behavior**
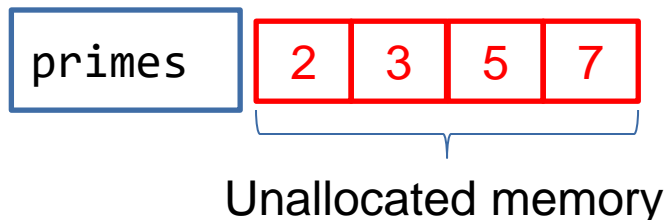
Function Templates

Goals:
- You can write your own function templates
- You can determine the concept of a template parameter
- You know the difference between templates and Java generics

OST
Ostschweizer
Fachhochschule

IFS INSTITUTE FOR SOFTWARE

- **Imagine you are tasked with implementing a function that returns the smaller of two given values**

- **For simplification: Just define that function for the fundamental types**

```cpp
int min(int left, int right) {
   return left < right ? left : right;
}
```

- **Multiple similar implementations just for covering all fundamental types!**

  - What about composite types? `std::string`

  - What about your own types? `Date, Word`?

- **Implementing an overload for every possible type leads to code duplication and is not maintainable**

  - Solution: Templates for compile-time polymorphism

```
template <Template-Parameter-List>
FunctionDefinition
```

- **Keyword `template` for declaring a template**

- **Template parameter list: Contains one or more template parameters**

  - ◼ A template parameter is a placeholder for a type, which can be used within the template as a type

  - ◼ A type template parameter is introduced with the `typename` (or `class`) keyword

- **Example**

Template Parameter

```
template <typename T>
T min(T left, T right) {
    return left < right ? left : right;
}
```

Template Definition

```
template <typename T>                                    min.h
T min(T left, T right) {
  return left < right ? left : right;
}
```

```cpp
#include "min.h"                                    smaller.cpp
#include <iostream>

int main() {
  int first;
  int second;
  if (std::cin >> first >> second) {
    auto const smaller = min(first, second);
    std::cout << "Smaller of " << first
              << " and " << second
              << " is: " << smaller << '\n';
  }
}
```

- **The compiler...**

  - ...resolves the function template

  - ...figures out the template argument(s)

  - ...instantiates the template for the arguments (creates code with template parameters replaced)

  - ...checks the types for correct usage

TemplateId

Template Instance

```
template <typename T>
T min(T left, T right) {
    return left < right ? left : right;
}
```

```
min(first, second)
```

min<int>

```
int min(int left, int right) {
    return left < right ? left : right;
}
```

- **Templates are usually defined in a header file**

  - A compiler needs to see the whole template definition to create an instance

  - Function template definitions are implicitly `inline`

- **The definition in a source file is possible, but then it can only be used in that translation unit**

- **Type checking happens twice**

  - When the template is defined: Only basic checks are performed: syntax and resolution of names that are independent of the template parameters

  - When the template is instantiated (used): The compiler checks whether the template arguments can be used as required by the template

- **Generics in Java require bounds defined by interfaces or types**

```java
public static <T extends Comparable<? super T>> T min(T first, T second) {
    return first.compareTo(second) < 0 ? first : second;
}
```

  - Only the capabilities specified by those bounds can be used

  - Type erasure removes the argument's type. It can only be instantiated via reflection.

- **C++ Templates use duck-typing**

```cpp
template <typename T>
T min(T left, T right) {
    return left < right ? left : right;
}
```

  - Every type can be used as argument as long as it supports the used operations

  - The actual type can still be used within a template, e.g. create an instance directly

- **Concept: The requirements a type must fulfill to be useable as an argument for a specific template parameter**

- **What are the requirements of the type `T` in our `min` function template?**

```
template <typename T>
T min(T left, T right) {
    return left < right ? left : right;
}
```

  - ◼ `<` Comparable with itself: `bool operator<(T, T)`

  - ◼ Copy/Move-Constructible, to return T by value

- **In C++20 it is possible to explicitly specify concepts**

  - ◼ Allows better checking of template definition (are all requirements fulfilled)

  - ◼ Better (easier to read) error messages for failed template instantiations

?

```cpp
template<class InputIt1, class InputIt2, class T>
T inner_product(InputIt1 first1, InputIt1 last1, InputIt2 first2, T init) {
    while (first1 != last1) {
        init = init + *first1 * *first2;
        ++first1;
        ++first2;
    }
    return init;
}
```

● **InputIt1/InputIt2:**

- ▪ * (Dereferenceable)

- ▪ ++ (Prefix increment)

- ▪ Only InputIt1:
  != with itself
  Result convertible to bool

● **init + *first1 * *first2:**

- ▪ * on *first1 and *first2

- ▪ + on T and result of above

● **T:**

- ▪ Assignable from result of
  init + *first1 * *first2

- ▪ Copy/Move Constructible

- **The compiler will try to figure out the function template's arguments from the call**

```
template <typename T>
T min(T left, T right) {
    //Implementation
}
```

```
min(1, 2);
```

Argument 1 has type `int`
Argument 2 has type `int`
Parameter 1 is T
Parameter 2 is T
T becomes `int`

- **Pattern matching on the function parameter list is used for deducing the correct argument**

```
template <typename T>
T min(std::vector<T> const & values) {
 //Implementation
}
```

```
std::vector<double> values{1.0, 2.0};
min(values);
```

Argument has type `std::vector<double>`
Parameter is `std::vector<T> const &`
T becomes `double`

```
template <typename T>
T min(T left, T right) {
    return left < right ? left : right;
}
```

```
min(1, 1.0)
```

?

```
int min(int left, int right) {
    return left < right ? left : right;
}
```

```
double min(double left, double right) {
    return left < right ? left : right;
}
```

- **In specific cases the number of template parameters might not be fix/known upfront**

  - ■ Thus the template shall take an arbitrary number of parameters

- **Example:**

```cpp
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
  std::cout << first;
  if (sizeof...(Types)) {
    std::cout << ", ";
  }
  printAll(rest...);
}
```

- **Syntax (ellipses everywhere): ...**

  - ■ `...` in template parameter list for an arbitrary number of template parameters (Template Parameter Pack)

  - ■ `...` in function parameter list for an arbitrary number of function arguments (Function Parameter Pack)

  - ■ `...` after `sizeof` to access the number of elements in template parameter pack

  - ■ `...` in the variadic template implementation after a pattern (Pack Expansion)

- **Template declaration:**

```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest);
```

- **Implicit instantiation:**

```
int i{42}; double d{1.25}; std::string book{"Lucid C++"};
printAll(i, d, book);
```

- **Template instance:**

```
void printAll(int const & first, double const & __rest0,
                                  std::string const & __rest1) {
  std::cout << first;
  if (2) { //sizeof...(Types) – Number of arguments in the pack
    std::cout << ", ";
  }
  printAll(__rest0, __rest1); //rest... expansion
}
```

- **`sizeof...(<PACK>)` will be replaced by the number of arguments in the pack parameter**

  - `0, 1, 2, ...`

```
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
 //...
 printAll(rest...);
}
```

- **Pattern: `rest`**

- **The pattern must contain at least one pack parameter**

- **An expansion is a coma-separated list of instances of the pattern**

- **For each argument in that pack an instance of the pattern is created**

- **In an instance of the pattern the parameter pack name is replaced by an argument of the pack**

```
void printAll(int const & first, double const & __rest0, std::string const & __rest1) {
  //...
  printAll(__rest0, __rest1); //rest...
}
```

- **For the call printAll(__rest0, __rest1): printAll<double, std::string>**

```cpp
void printAll(double const & first, std::string const & __rest0) {
  std::cout << first;
  if (1) { //sizeof...(Types) – Number of arguments in the pack
    std::cout << ", ";
  }
  printAll(__rest0); //rest... expansion
}
```

- **For the call printAll(<rest0>): printAll<std::string>**

```cpp
void printAll(std::string const & first) {
  std::cout << first;
  if (0) { //sizeof...(Types) – Number of arguments in the pack
    std::cout << ", ";
  }
  printAll(); //rest... expansion
}
```

- **What about printAll()?**

- **What about `printAll()`?**

  - ■ The variadic template `printAll` is not viable, as it requires at least one parameter

- **We need a base case for the recursion**

```cpp
void printAll() {
}
```

- **Wouldn't it be feasible to just rearange the code in the variadic template?**

```cpp
template<typename First, typename...Types>
void printAll(First const & first, Types const &...rest) {
  std::cout << first;
  if (sizeof...(Types)) {
    std::cout << ", ";
    printAll(rest...);
  }
}
```
?

?

```cpp
template <typename F, typename...T>
F min(F const & first, T const &...rest) {
  auto const & restMin = min(rest...);
  return first < restMin ? first : restMin;
}

int main() {
  std::cout << min(3, 1, 4, 1, 5);
}
```

Incorrect

The recursive min function template lacks a base case. The call `min()` will not be resolvable. Furthermore, it would be difficult to specify that overload, because it would need to return a value that is smaller than every other value of the given type.
Solution: overload `min(F const & first)`

```cpp
template <typename F, typename...T>
F min(F const & first, T const &...rest) {
  auto const & restMin = min(rest...);
  return first < restMin ? first : restMin;
}
```

Correct

Template code that is not used will not be instantiated. Therefore, the missing overload `min()` will not be recognized.

- **String literals as arguments**

```cpp
int main() {
  std::cout << min("Gregor Clegane", "Tyrion Lannister");
}
```

- **Possible output: Tyrion Lannister**

min<char const *>

```cpp
template <typename T>
T min(T left, T right) {
  return left < right ? left : right;
}
```

➡️

```cpp
char const * min(char const * left,
                 char const * right) {
  return left < right ? left : right;
}
```

- **Comparison happens based on the pointer (address) of the arguments**

- **Multiple function templates with the same name can exist**

  - As long as they can be distinguished by their parameter list

- **An overload for pointers is possible**

```cpp
template <typename T>
T min(T left, T right) {
  return left < right ? left : right;
}

template <typename T>
T * min(T * left, T * right) {
  return *left < *right ? left : right;
}
```

```
template <typename T>
T min(T left, T right) {
  return left < right ? left : right;
}

template <typename T>
T * min(T * left, T * right) {
  return *left < *right ? left : right;
}
```

- **Will this help for the string literal example?**

```
std::cout << min("Gregor Clegane", "Tyrion Lannister");
```

- **And this?**

```
std::cout << min("Samwell Tarly", "Sansa Stark");
```

● **Function templates and "normal" functions with the same name can coexist**

```cpp
template <typename T>
T min(T left, T right) {
  return left < right ? left : right;
}

template <typename T>
T * min(T * left, T * right) {
  return *left < *right ? left : right;
}

char const * min(char const * left,
                 char const * right) {
  return std::string{left} < std::string{right} ? left : right;
}
```

- **Operators and member functions can be templates too**

- **Example**

```cpp
auto const printer = [&out](auto const & e) {
  out << "Element: " << e;
};
```

```cpp
struct __PrinterLambda {
  template <typename T>
  void operator()(T const & e) const {
    __out << "Element: " << e;
  }
  std::ostream & __out;
};
```

- **Beware: Don't make operator templates too eagerly, you might end up with unexpected (better) matches for other calls.**

- **Literals and references**

```cpp
template <typename T>
T const & min(T const & left, T const & right) {
    return left < right ? left : right;
}
```

```cpp
std::cout << min("C++", "Java");
```

```
error: no matching function for call to
'min(const char [4], const char [5])'
```

- **Literal suffix helps**

```cpp
using namespace std::string_literals;
std::cout << min("C++"s, "Java"s);
```

- **Templates might be a better match**

  - For example because of constness

```cpp
template <typename T>
T min(T & left, T & right) {
  return left < right ? left : right;
}

std::string min(std::string const & left, std::string const & right) {
  return lexicographical_compare(begin(left), end(left),
                                 begin(right), end(right),
                                 [](char l, char r) {
    return tolower(l) < tolower(r);
  }) ? left : right;
}
```

```cpp
std::string small{"aa"};
std::string capital{"ZZ"};
std::cout << min(small, capital) << '\n'; //ZZ
```

- **Temporary might become invalid**

```
template <typename T>
T const & min(T const & left, T const & right) {
  return left < right ? left : right;
}
```

```
std::string const & smaller = min("a"s, "b"s);
std::cout << "smaller is: " << smaller;
```

- **Lifetime of temporaries ends at ;**

- **const & can extend the lifetime of a temporary, but only if it is a temporary value as a result of the outermost expression**

?

```
template <typename T>
T const &
min(T const & left, T const & right) {
  return left < right ? left : right;
}

int main() {
  std::cout << min("Java", "Rust");
}
```

Incorrect

This code actually compiles, as both string literals have type char const[5]. However, the comparison will still happen on the effective addresses of the arrays.

```
template <typename T>
T const &
min(T const & left, T const & right) {
  return left < right ? left : right;
}

int main() {
  std::string java{"Java"};
  std::string const rust{"Rust"};
  std::cout << min(java, rust);
}
```

Correct

Even though one string is const and the other is not, the non-const string can be passed as an argument to a const reference.

- **Function templates provide way to specify a set of functions for types that behave in a certain way**

- **Arguments for a function template instance might be deduced by the compiler**

- **Variadic templates take an arbitrary number of arguments**

- **Template instantiation happens at compile-time**